Санкт-Петербургский государственный университет

Алексеев Евгений Григорьевич

Выпускная квалификационная работа бакалавра

Эвристический параллельный алгоритм поиска оптимизирующего маршрута на связном графе

Направление 02.03.02.

«Фундаментальная информатика и информационные технологии»

ООП «Программирование и информационные технологии»

Научный руководитель,

кандидат

физико-математических наук

Брэгман К.М.

Санкт-Петербург

2018

Содержание

Введение

Актуальность работы

В настоящее время поиск лучшего маршрута для путешествий - актуальная тема почти для каждого современного человека. Всё чаще люди пользуются различными информационными системами для туризма и путешествий. Интересных мест в больших городах огромное количество, а способов посетить их все - еще больше. Но может случиться так, что выбранный порядок посещений окажется крайне неэффективным: удастся посетить лишь пару мест, да и те окажутся скучными и не стоящими внимания. В такие моменты на помощь приходят популярные сервисы-путеводники, веб-сайты с их персональными гидами по городу, но все они несовершенны: не позволяют построить маршрут, учитывающий необходимые ограничения, или же не позволяют увидеть все возможные варианты. Для того чтобы решить описанные выше проблемы постоянно разрабатываются новые вспомогательные информационные системы на основе различных алгоритмов, в зависимости от нужды пользователей. При разработке подобных систем со стороны разработчиков встаёт вопрос об организации архитектуры таким образом, чтобы необходимые расчеты производились быстро и без непозволительных трудозатрат.

Представленный в данной работе алгоритм поиска оптимизирующего маршрута позволит реализовать разработчикам собственные проекты под нужды пользователей и в результате туристам и гостям города больше не нужно будет тратить время на составление удобного маршрута.

Результатом этой работы стал комплекс программ позволяющий определить и визуализировать наиболее “интересный” маршрут на графе.

Цель работы

Основной целью данной работы является создание и реализация эффективного алгоритма поиска, способного обрабатывать входные данные и выдавать результат за приемлемое время. Также должен быть проведен анализ схожих задач и подходов к их решению, для составления наиболее полной картины в данной сфере. В дополнение должна быть сформирована архитектура программного комплекса, позволяющая в короткие сроки и без чрезмерных усилий встроить реализованный алгоритм в свою структуру, не закрывая при этом доступа к модификации и расширению возможностей.

Теория графов

Формулировка нашей задачи содержит в себе термины из теории графов, а алгоритм был разработан с целью решения вычислительных проблем которые могут быть сведены к поиску нужных путей в графах. Следовательно, базовые знания из теории графов помогут понять как проблему, так и ее решение. Далее будут описаны несколько основных определений. Следующие определения взяты из [1].

- Конечный граф G состоит из двух конечных множеств *V* и *E*. Элемент *v* из *V* называется вершиной (узлом), а элемент *e* из *E* называется ребром. Если необходимо указать принадлежность вершины или ребра к графу пишется V(G) или E(G) соответственно.
- Маршрутом (путем) в графе называется чередующаяся последовательность вершин и ребер v_0, $e_{0,1}$, v_1, …, $e_{k-1,k}$, v_k, в которой любые два соседних элемента инцидентны. Если $v_0 = v_k$, то маршрут замкнут, иначе открыт. В условиях нашей задачи рассматривается поиск именно замкнутого маршрута, который будет обозначаться *P*.
- Путь называется простым, если не проходит дважды через одно ребро.
- Граф называется связным, если между каждой парой вершин существует как минимум один путь.
- Полный граф — граф, в котором для каждой пары вершин v_1 v_2, существует ребро, инцидентное v_1 и инцидентное v_2 (каждая вершина соединена ребром с любой другой вершиной).

Постановка задачи

Дан произвольный связный граф, не обязательно полный. Для каждой вершины $v_i \in V = \{v_1, v_2, \ldots, v_n\}$, где n - количество вершин в графе, указан вес $w_i \in R$, означающий интерес вершины. Для каждого ребра $e(v_i, v_j)$, соединяющего вершины $[v_i;v_j]$, есть время пути $t_{ij} > 0$ и полагается, что $t_{ij} = t_{ji}$.

$L(P)$ - количество вершин в пути P.

Время пути P обозначается как T(P) и находится как сумма всех ребер маршрута:

$$T(P) = \sum_{i=0}^{L(P)-1} t(P_i, P_{i+1}), \quad (1)$$

где L(P) - длина маршрута.

Аналогично определяется интерес маршрута как сумма уникальных вершин:

$$I(P) = \sum_{i=0}^{L(P)} w(P_i) * \delta(P_i), \quad (2)$$

где L(P) - длина маршрута, а

$$\delta(P_i) = \begin{cases} 1, & \text{если } w_i \text{ не учтено в сумме} \\ 0, & \text{если } w_i \text{ учтено в сумме} \end{cases} \quad (3)$$

Необходимо построить маршрут $P = \{v_{0,}, \ldots, v_i\}$, где $v_i = v_0$ из стартовой позиции v_0 в конечную, не превышающий по времени заданного ограничения T_{max} и имеющий максимальный интерес из возможных в текущих условиях, то есть такой путь P, что:

$$I(P) \rightarrow max, \text{ при этом } T(P) <= T_{max} \quad (4)$$

Вершины могут повторяться, но их интерес не учитывается повторно в текущем пути, в отличии от ребер ведущих в них.

Сравнение с типовыми задачами

Следующие задачи имеют схожую постановку, но отличаются условиями, вследствие чего, решения, подходящие для них, не подходят для нашей задачи, но принципы, использованные в них, могут помочь при создании собственного решения.

Задача коммивояжера

Заключается в отыскании самого выгодного маршрута, проходящего через указанные города хотя бы по одному разу с последующим возвратом в исходный город [2]. Но в нашей задаче дополнительно присутствуют ограничение на время, а вершины имеют своей вес, и неважно, сколько раз мы будем проходить через вершины. Таким образом решения задачи коммивояжера не полностью удовлетворяют условиям нашей задачи.

Задача о доставке грузов с временными интервалами

Суть задачи состоит в следующем: предположим, дано множество вершин (узлов) $N = \{n_0, n_1, n_2, ..., n_m\}$, где n_0 означает стартовую позицию, а каждая n_i , i = 1, 2, … m означает местоположение клиента, причем m - чётно. Так как для каждого клиента есть пара точек подбора и доставки, можно предположить, без потери общности, что множество $\{n_0, ..., n_{m/2}\}$ отражает местоположение точек подбора, а множество $\{n_{(m/2)+1}, ..., n_m\}$ отражает местоположение точек доставки, так, что точка подбора n_i имеет соответствующую точку доставки $n_{i+(m/2)}$. Каждая позиция n_i, $i \neq 0$ связана с:

- Спросом клиента q_i , таким, что $q_i > 0$ для точки подбора, $q_i < 0$ для точки доставки и $q_i + q_j = 0$ для точек подбора и доставки одного и того же клиента.
- Временным окном (Time Window) $[e_i , l_i]$ в течении которого позиция должна быть обработана и $l_i > e_i$.

  Для каждого возможного ребра $< n_i , n_j >$ время пути t_{ij} указано и

предполагается, что $t_{ij} = t_{ji}$.

Есть машина с ограниченной вместимостью C. Ограничение на вместимость необходимо для того, чтобы груз перевозимый машиной не превышал ее вместимости.

Машина начинает свой путь в стартовой позиции и заканчивает в любой точке доставки. Каждая точка должна быть посещена один раз.

Каждая позиция должна быть обслужена в течение временного окна, например, если машина подъедет к указанной точке до самого раннего времени обслуживания e_i, то ей придется подождать. Ограничение предшественника (Precedence constraint) требует, чтобы каждая точка подбора предшествовала точке доставки [3].

Можно провести аналогии с нашей задачей: если взять временное окно длинной в ограничение по времени $l_i - e_i = T_{max}$, точкам подбора установить в соответствие вершины графа, а точки доставки установить в стартовой позиции, мы получим условия, схожие с нашей задачей, но не удовлетворяющие таким необходимым требованиям, как максимизация интереса - из-за отсутствия веса у вершин. В данной задаче существует аналогия с весом вершин - точки у которых истекает временное окно имеют больший приоритет, чем точки у которых время обслуживание еще не подошло или имеет достаточный интервал, но это значение будет являться динамическим, в то время как в нашей задаче значения интереса статичны.

Поиск кратчайших путей

Является частью нашей задачи, но не накладывает ограничений на искомые пути. Несомненно, при поиске оптимизирующего маршрута необходимо будет использовать эффективные алгоритмы поиска кратчайших путей. Так, например, алгоритм Беллмана-Форда [4] используется для предобработки исходного графа и нахождения кратчайших путей от вершины к вершине.

Способы решения

Рассмотрим способы решения каждой из описанных выше задач

Алгоритм имитации отжига (Simulated annealing)

Один из примеров методов Монте-Карло основывается на физическом процессе, который происходит при кристаллизации вещества, в том числе при отжиге металлов. При медленном остывании металлов, после значительного нагрева атомы стремятся выстроиться в кристаллическую решетку с минимальной энергией. Переход атома из одной ячейки в другую происходит с некоторой вероятностью, причём вероятность уменьшается с понижением температуры. Для использования этого алгоритма в задаче коммивояжера необходимо определить:

- функцию энергии (целевая функция, которую мы оптимизируем);
- убывающую функцию изменения “температуры”;
- функцию, порождающую новое состояние.

Функция изменения “температуры” определяет время работы алгоритма и ставит в соответствие с номером операции некоторое число-температуру.

Функция, порождающая новое состояние, позволяет получить новое состояние-кандидат, в которое система может перейти, а может и отбросить. К примеру, в задаче коммивояжера это может быть реализовано как функция возвращающая путь, в котором между двумя случайными вершинами путь был инвертирован (рис. 1).

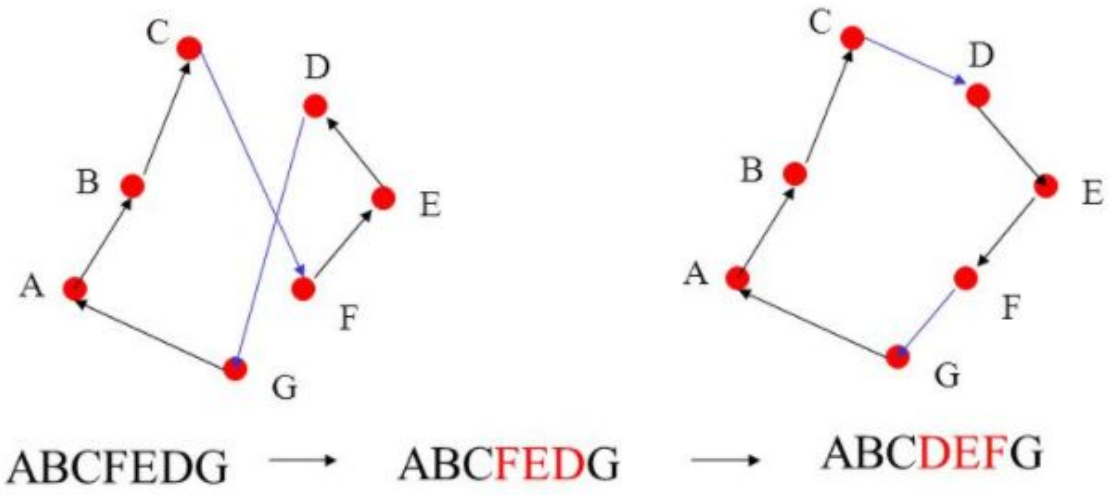

рис.1

Подробнее использование данного алгоритма описывается в [5].

Симуляция отжига - это вероятностный алгоритм, который не гарантирует нахождение оптимального маршрута. Критичным для нас условием является полнота графа, то есть доступность из каждой вершины каждой. Так как в условиях города такое условие выполняется крайне редко, подобные вероятностные алгоритмы не могут использоваться для нашей задачи.

Поиск с запретом (Tabu Search)

Tabu Search - это мета-алгоритм, позволяющий локальному эвристическому поиску исследовать пространство решений за пределами локальных оптимумов.

Так же, как и метод имитации отжига, Tabu Search можно рассматривать как улучшающий метод [8]. Это такие схемы поиска, в которых рассматриваются лучшие соседи текущего решения, а целевая функция может ухудшиться, если не может улучшиться. Таким образом, избегаются локальные оптимумы и достигается глобальный. Как правило,

подобные алгоритмы способны останавливаться в любой момент, а их время работы, которое может быть довольно большим, не зависит полиномиально от размера входных данных.

Основы TS могут быть описаны следующим образом. Дана функция *f(x)*, которую необходимо оптимизировать на множестве X, алгоритм начинает так же, как обычный локальный поиск, итеративно проходя от одной точки (решения) до другой, пока не будет достигнут критерий останова. Каждый $x \in X$ имеет множество соседей $N(x) \subset X$, и каждое решение $x \in N(x)$ достижимо из x с помощью операции называемой "*move*".

TS выходит за рамки локального поиска, реализуя стратегию изменения N(x) по ходу поиска, эффективно изменяя его на другое множество N*(x).

Решения N*(x) определяются несколькими способами. Один из них, дающий алгоритму "Tabu Search" его имя, идентифицирует решения на определенном промежутке, и запрещает их дальнейший выбор в множестве N*(x), называя их tabu (запретом). Например, в некоторых реализациях алгоритма могут запрещать ту часть решения, которая изменялась на последних итерациях.

Ant colony optimization (ACO)

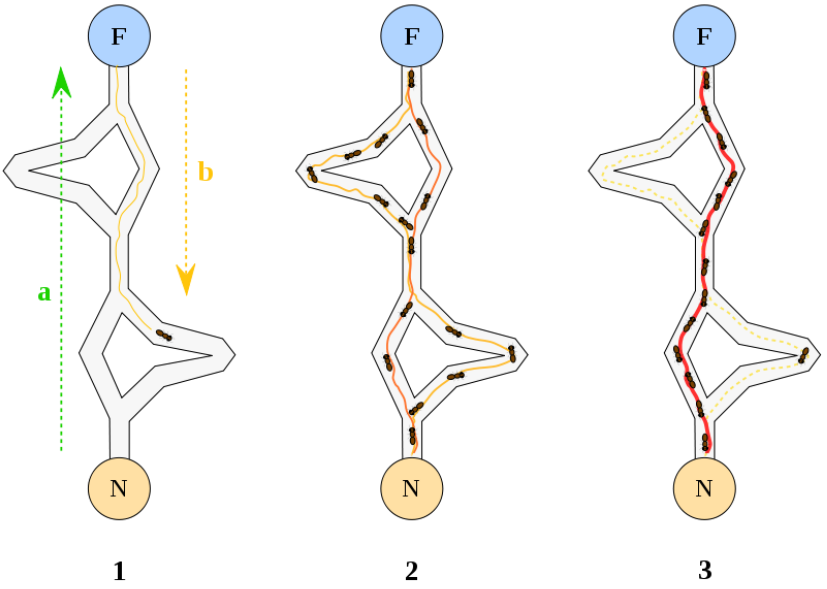

рис. 2

Муравьиные алгоритмы широко используются в методах оптимизации и представляют собой заимствование другого реального явления - коллективный разум муравьиной колонии. Оригинальная идея исходит от наблюдения за муравьями в процессе поиска кратчайшего пути от колонии до источника питания. Муравьи изначально ходят в случайном порядке и при нахождении продовольствия (F) возвращаются в гнездо (N) (рис. 2.1). Последующие муравьи проходят всевозможными путями к источнику пищи, оставляя за собой след из феромонов (рис. 2.2), при этом в выборе пути, с долей вероятности отдавая предпочтение путям с более сильным следом от предыдущих муравьев. Так как с длинных маршрутов след будет испаряться быстрее, со временем все муравьи начнут ходить по кратчайшему пути, а неэффективные маршруты исчезнут из-за испарения феромонов (рис. 2.3). Таким образом, ACO может применятся для поиска кратчайших путей в графе. Подробнее муравьиные алгоритмы описаны в [11].

Алгоритм решения

Поиск оптимизирующего маршрута P связан ограничением по времени T_{max} и необходимостью максимизировать $I(P)$, поэтому существующие алгоритмы требуют значительной переработки для достижения необходимого результата. Поскольку эта задача настолько же трудоемкая, насколько и разработка собственного алгоритма, было решено разработать новый алгоритм с оглядкой на существующие.

В связи с тем, что отсутствует гарантированный способ определить, не было ли пропущено правильное решение, высокая скорость работы алгоритма достигается за счет удаления дублирующих путей, а также за счет исключения маршрутов, которые являются заведомо неоптимальными или содержат неэффективные переходы между вершинами.

Использованные средства

Работа была выполнена на платформе .NET. Класс ThreadPool, как инструмент для реализации эффективной многопоточности - подробнее в [6]. Визуализация и взаимодействие между компонентами выполнены с помощью платформы Unity. Unity - это межплатформенная среда разработки, позволяющая создавать приложения работающие под различными операционными системами. Данная среда была выбрана из-за простоты разработки, компонентного подхода к архитектуре проектов и слабой связностью с системой под которую изначально ведется разработка. Также решающим фактором, повлиявшим на выбор данной платформы была скорость разработки прототипа: описанные выше особенности системы позволяют в кратчайшие сроки получить видимый результат без весомых трудозатрат. Все эти плюсы позволили реализовать алгоритм таким образом, что его дальнейшее использование в иных системах не вызовет затруднений.

Эвристика

Алгоритм называется эвристическим, если в нем применяются определенные практические методы (эвристика), не являющиеся гарантированно точными или оптимальными, но достаточные для решения поставленной задачи. В текущем решении была выбрана эвристика вида

$$H(V_i , V_j) = W_j / t_{ij} , \qquad (4)$$

где j - индекс вершины, в которой рассчитывается текущая эвристика,

W_j - интерес вершины j , t_{ij} - вес ребра между вершинами V_i и V_j.

Подобная эвристика позволяет адекватно оценить “желательность” вершины в текущем пути.

Для улучшения работы алгоритма рассчитывается наилучшая возможная эвристика H_{max} для каждой вершины. Она вычисляется как

наибольшая из сумм эвристик между двумя смежными вершинами состоящих в кратчайшем пути до текущей вершины.

Эвристика в алгоритме используется при выборе новых вершин в текущий путь. Вершинам с наибольшим значением эвристики отдается приоритет, таким образом они будут просчитаны первее, а, следовательно, путь через вершины с меньшим шансом на наличие искомых оптимальных маршрутов будут отсеиваться на обратном пути, не доходя до конца (См. “Второй этап”).

Псевдокод распределения эвристики по графу:

```
Дано:
   Рассчитана глубина вершин считая от стартовой любым алгоритмом.
   MaxDepth — максимальная глубина графа
   Вершины графа — V
   Ребра графа
   H(Vi , Vj)- функция возвращающая значение эвристики из Vi в Vj
   Hmax(V) - лучшая эвристика вершины набранная на текущей итерации
на старте равная 0.
Результат: правильно рассчитанная Hmax в каждой необходимой вершине
for i := 0; i < MaxDepth; i ++
   foreach V1 ∈ V где глубина == i
         foreach V2 ∈ смежные с V1 и глубиной == i
               if (Hmax(V1) < Hmax(V2))
                     if (H(V2 , V1) + Hmax(V2) > Hmax(V1))
                           Hmax(V1) :=  H(V2 , V1) + Hmax(V2)
                     End
               else
                     if (H(V1 , V2) + Hmax(V1) > Hmax(V2))
                           Hmax(V2) :=  H(V1 , V2) + Hmax(V1)
                     End
               End
         End
         foreach V2 ∈ смежные с V1 и глубиной > i
               if(H(V1 , V2) + Hmax(V1) > Hmax(V2))
                     Hmax(V2) :=  H(V1 , V2) + Hmax(V1)
               End
         End
```

```
        End
    End
```

Таким образом, после выполнения описанного алгоритма мы получим значение BM в каждой вершине графа и сможем руководствоваться этим значением при выборе следующей вершины или ветви графа.

Параллельность

Рекурсивные алгоритмы с множеством вызовов отлично поддаются распараллеливанию: каждая ветвь решения может производить вычисления параллельно и независимо от остальных ветвей. Единственная пересекающаяся информация, необходимая ветвям, - это текущий лучший путь, и информация о состоянии вершин - так как каждая из них хранит в себе лучший интерес, набранный на обратном пути.

Для реализации распараллеливания алгоритма был выбран класс ThreadPool [6], предоставляющий пул потоков, который можно использовать для выполнения задач, отправки рабочих элементов, обработки асинхронного ввода-вывода, ожидания от имени других потоков и обработки таймеров.

Известно, что создание, уничтожение, переключение между потоками – это дорогостоящие операции. Для того, чтобы избежать накладных расходов, связанных с этим, основной идеей пула потоков в .NET стало уменьшение числа задействованных потоков и увеличение выполненной ими работы.

Не углубляясь в детали, пул потоков реализован как потокобезопасная рабочая очередь с группой потоков-работников, позволяющая запускать “Работы” (ветви решений в нашем случае) и исключающая простой “работников”, выдавая им задачу из очереди, как только они закончат свою работу.

Количество рабочих потоков по умолчанию - 250 единиц на CPU и

1000 Input/Output потоков на процесс, но эти значения можно изменить вызовом соответствующих методов.

Таким образом, для эффективного использования многопоточности и сокращения времени простоя потоков во время ожидания освобождения общих ресурсов и распределения задач было решено запускать ветвь решения в отдельном потоке только из стартовой позиции.

В каждой ветви работает последовательный алгоритм, но при этом он имеет постоянно обновляющуюся информацию из других потоков, что позволяет сократить количество обрабатываемых вариантов и получить прирост в скорости работы алгоритма.

Реализация

Архитектура программного комплекса

Для получения результата основная программа должна получить на вход заранее подготовленный граф. Такой граф представляет собой обработанные, разобранные на составляющие, данные с карт. Их данные об улицах можно получить с любого сервиса карт, предоставляющего такую возможность. Также граф можно представить в виде структурной схемы в текстовом формате, например, JSON или XML, который предоставляется человеку в читаемом виде и позволяет легко вносить правки в данные, а перенос и сохранение информации не отличается по сложности реализации от загрузки.

Далее данные проходят через предобработку, получая недостающую для работы основного алгоритма информацию о графе, и в работу вступает алгоритм поиска нужного нам маршрута. Алгоритм состоит из двух этапов, каждый из которых несет в себе определенные цели. Первым этапом составляется новый путь, в который на каждой итерации добавляется по

вершине. Каждая ветвь алгоритма соотносится с определенным путем и, при достижении установленных условий, может перейти во второй этап работы.

Если ветвь первого этапа не оборвалась (то есть не стала тупиковой, не приводящей к результату), путь из этой ветви передается во второй этап, который может выдать конечный результат-маршрут, возможно, лучший чем предыдущий.

Путь, превзошедший текущий лучший маршрут, визуализируется, заменяя последний, обновляется статистика.

Рассмотрим подробнее каждый этап:

Предобработка

- Расчет минимальных расстояний $D_{min}(V_i)$ от стартовой позиции до каждой вершины, например алгоритмом Беллмана-Форда [7].
- Расчет глубины для каждой вершины из конечной и начальной вершин.
- Инициализация потоков и визуализации.
- Расчет эвристической характеристики для каждой вершины графа.
- Расчет статистики текущего графа - максимальный интерес, время и прочее.

Первый этап - “Поиск”

Для каждой смежной вершины мы выбираем ветвь, наиболее подходящую по эвристике (по весу вершины, если функция эвристики не используется в текущем расчете), и набираем в путь вершины до тех пор, пока не достигнем точки, при которой добавление новых вершин приведет к нарушению ограничений по времени или исчерпанию возможных вершин. Таким образом, если при рассмотрении добавления в путь P новой вершины V_{next}, с последней добавленной вершиной V_{last}, получаем, что если

$$T(P) + t_{last,\ next} + D_{min}(V_{next}) > T_{max}, \qquad (5)$$

то вершина V_{next} пропускается, а текущий путь помечается меткой “BackpathNeeded”, по значению которой определяется, нужен ли вызов второго этапа алгоритма. Таким образом, не будут пропущены возможные оптимальные маршруты.

Псевдокод первого этапа:

```
Название метода : Search(Path)
Дано:
      Path_last - последняя добавленная в путь вершина
      Start - стартовая позиция
      Метод AllNeighboursUsed() - возвращающий true если все смежные с
      текущей вершины содержатся в Path
      Метод CheckConditions(V_next) - возвращающий true если при добавлении
      вершины V_next нарушаются условия описанные выше
      Метод BackPath(Path) - описан ниже.
      Path.Add(V) - добавляет вершину в путь.
      return - незамедлительно завершает выполнение метода.
if(AllNeighboursUsed())
      if(Path_last  == Start)
            BackPath(Path)
            return
      End
      V_next := смежная с текущей вершина в которой путь до Start минимален
      Search(Path.Add(V_next))
      return
End
BackpathNeeded := false
for V_next  ∈ смежные с Path_last и упорядоченные по убыванию ВМ
      if (CheckConditions(V_next))
            if(Path_last == Start)
                  continue
            End
            BackpathNeeded := true
            continue
      End
      Search(Path.Add(V_next))
End
```

```
if(BackpathNeeded  == true)
      BackPath(Path)
End
```

Второй этап - “Возврат”

Используется рекурсивный принцип первого этапа, но вершины V_{next} нарушающие условия задачи не используются. Если выполняется

$$T(P) + t_{last,\ next} + D_{min}(V_{next}) > T_{max} \qquad (6)$$

вершина V_{next} будет отброшена.

На данном этапе в каждой вершине обновляется текущий лучший путь BP, который проходил через эту вершину, чтобы последующие ветви с путями P отсеивались, если I(BP) > I(P) или I(BP) == I(P), но при этом T(BP) < T(P), из-за последнего условия минимизируется время найденных маршрутов.

Текущий лучший путь обновляется, если Path лучше текущего по интересу.

Когда алгоритм проходит текущий этап, он останавливается в стартовой вершине, даже если есть возможность составить лучший путь.

Псевдокод второго этапа:

```
Название метода BackPath(Path)
Дано:
      Path_last - последняя добавленная в путь вершина
      Start - стартовая позиция
      Метод CheckBranch - возвращающий true если Path хуже значений
      записанных в текущей вершине.
      Path.Add(V) - добавляет вершину в путь.
      return - незамедлительно завершает выполнение метода.
      Метод CheckConditions(V_next) - возвращающий true если при добавлении
      вершины V_next нарушаются условия описанные выше
if(CheckBranch)
      return
End
BP = Path
if(Path_last == Start)
      return
```

```
End
for V_next ∈ смежные с Path_last
      if( CheckConditions(V_next))
            continue
      End
      BackPath(Path.Add(V_next)
End
```

Временная сложность

Важным критерием качества алгоритма является его временная сложность. Необходимо понять, насколько его применение оправдано и работает ли он эффективно. Для этого определяются количество элементарных операций, совершаемых алгоритмом, а в случаях рекурсивных алгоритмов необходимо также учитывать и количество вызовов функции. Временную сложность алгоритмов можно классифицировать в три категории:

- Сложность в худшем случае — когда время, необходимое алгоритму при текущих входных данных, **максимально**
- Средняя сложность — необходимое алгоритму время **среднее**
- Сложность в лучшем случае — когда время, необходимое алгоритму, **минимально**

Введем некоторые понятия, которые могут нам пригодиться.

n - число вершин в графе

m - число ребер в графе

$V_{good} \in V$ - вершины, которые удовлетворяют условиям

E - количество ребер

E_i - количество ребер, смежных с вершиной V_i

Последовательный вариант алгоритма

- Лучший случай:

Лучшим случаем можно назвать ситуацию, при которой в первом же предложенном пути было набрано максимум интереса, то есть были посещены все вершины графа, тогда другие пути искать нет необходимости, и алгоритм завершит свою работу, а оценка будет **O(n)**

- Худшим случаем будет являться полный граф, в котором ограничение на время достаточно большое, чтобы пройти все вершины графа, кроме одной, но недостаточное для того, чтобы получить замкнутый маршрут, включающий в себя все вершины. Таким образом не будет достигаться ни одно из условий остановки, и необходимо будет проверить все возможные пути, так как не представляется возможным определить, является ли текущий путь оптимальным и не был ли пропущен лучший, и придется рассматривать все возможные вершины. Сложность алгоритма в данном случае будет **O(n!)**, что при больших размерах данных даст неприемлемое время выполнения. Для того, чтобы не попадать в худший случай необходима предобработка входных данных, позволяющая сократить их объем и, следовательно, время работы алгоритма.
- Средний случай - лучший и худший случаи характеризуются моментом, в который был достигнут какой-либо из критериев останова: в лучшем случае они достигаются при первом же проходе графа, а в худшем - не достигаются до самого конца. Средним будет считаться случай, в котором один из критериев будет достигнут примерно на половине обхода графа.

Каждому произвольному графу $G_n \in G$ приписывается некоторая вероятность срабатывания условия остановки $P(G_n)$. ($P(G) \in [0; 1], \sum_{Gn \in G} P(G_n) = 1$). Для лучших случаев $P(G_{best}) = 1$ а для худших $P(G_{worst}) = 0$.

Обозначим функцией затрат времени нашего алгоритма от входного графа $G^* \in G$, как $C^T(G^*)$, которая будет являться случайной величиной на заданном выше вероятностном пространстве.

Средней сложностью называется математическое ожидание соответствующей случайной величины:

$$T_{ср.} = \sum_{i=0}^{n} P(G_i)C^T(G_i) \quad (7)$$

Так как в рамках нашей задачи не предоставляется возможным определить распределение входных данных (каждый город разный и в основном имеет уникальную структуру) анализ средней сложности остается на плечах дальнейших разработчиков, обладающих большей статистикой и реальными данными.

Параллельный вариант алгоритма

Вследствие того, что в каждом потоке запускается последовательный алгоритм, ускорение, получаемое благодаря использованию многопоточности, равно количеству потоков, способных работать одновременно.

Чем больше будет потоков, тем больше будет вероятность простаивания потоков в режиме ожидания разблокировки общих ресурсов. Таким образом, увеличение до максимума количества потоков не приведет к максимальному ускорению. Необходим компромисс, например в нашем алгоритме новые потоки создаются только из стартовой вершины, параллельная обработка вершин графа повышает вероятность нахождения подходящего пути и повышает скорость обновления значения BP (лучшего пути) в вершине, что обеспечивает повышение скорости за счет отбрасывания заведомо худших, чем уже найденные, ветвей.

Следовательно, ускорение параллельного варианта будет равно времени работы последовательного алгоритма, деленного на количество

смежных вершин k:

$$T^*(n) = T_1(n) / k \qquad (8)$$

Но, так как невозможно избежать подготовительного этапа при работе с потоками, а использование множества потоков ведет к неизбежным простоям в блокировках, это не будет окончательной временной оценкой.

Необходимо учитывать подготовительный этап, зависящий от возможностей системы, на которой работает алгоритм, и время простоя в блокировках, отсутствующее в последовательном варианте алгоритма:

$$T(n) = T^*(n) + T_{п.} + T_{ож.} \qquad (9)$$

где: $T_п$ - время, необходимое для подготовки потоков и алгоритма к работе; $T_{ож.}$ - время простоя алгоритма в блокировке, зависящее сугубо от структуры графа и управления потоками.

Тесты

Тестирование алгоритмов является неотъемлемой частью разработки, и далее будут приведены таблицы, содержащие результаты отработки алгоритма на массиве случайно сгенерированных графов произвольной, удовлетворяющей условиям нашей задачи, структуры. Для каждой вероятности проводилось десять тестирований с ограничением по времени в 40 единиц. Замеры времени на каждом графе проводились в последовательных (послед.) и параллельных (параллел.) режимах.

Необходимо также учитывать, что тестирование производилось не на предназначенном для этого оборудовании, а на персональном компьютере.

Генерация графа происходит по простейшей, прямолинейной схеме: в зависимости от установленной вероятности генерируются вершины, а затем между ними устанавливаются связи (ребра).

Для тестов максимально возможное количество вершин было

установлено на отметке 200, следовательно, для каждой позиции в зависимости от вероятности появления определялось, будет там вершина или нет.

Тесты в Unity с визуализацией

Вероятность появления вершины	Среднее время работы сек.	Максимальное время работы сек.	Минимальное время работы сек.	Среднее количество итераций алгоритма
0,9	0.1029 / 0.0575	0.704 / 0.406	0.004 / 0.003	658
0,7	0,2778 / 0,220	1.778 / 1.26	0.038 / 0.023	2901
0,5	1,0393 / 0,6992	4.7/3.6	0.029 / 0.024	11594
0.3	4,282 / 3,7133	13.975 / 14.5	0.192 / 0.101	57877
0.1	6,1165 / 4,3652	17.67 / 11.25	0.074 / 0.064	173737

(Через "/" указаны измерения для последовательного и параллельного вариантов, подробные данные см. в Приложении)

По результатам тестов становится очевидной зависимость работы алгоритма от качества входных данных. Обработанные заранее графы, в которых убраны излишние вершины и ребра, позволяет производить меньше вычислений, а время работы алгоритма значительно сокращается.

Так как визуализация занимает некоторое время, а работа алгоритма через Unity медленнее чем без, ниже приводятся замеры времени алгоритма на чистом .NET.

Работа алгоритма без визуализации

Вероятность появления вершины	Среднее время работы сек. (послед./параллел.)	Максимальное время работы сек. (послед./параллел.)	Минимальное время работы сек. (послед./параллел.)
0,9	0.511 / 0.0232	0.332 / 0.215	0.003 / 0.001
0,7	0.17249 / 0,1312	0.878 / 0.532	0.021 / 0.017
0,5	0.523 / 0.3291	2.83 / 1.23	0.018 / 0.013

0.3	2.187 / 1.693	6.675 / 6.31	0.093 / 0.082
0.1	4.2138 / 2.2234	10.235 / 6.79	0.032 / 0.054

(Через “/” указаны измерения для последовательного и параллельного вариантов)

Исходя из данных видно, что на не специализированном устройстве визуализация занимает значительное время, и алгоритм выполняется примерно в два раза быстрее без нее.

Таким образом, очевидно, что для вычислительно сложных задач лучше не использовать не предназначенные для этого платформы, но сугубо в показательных целях и ради скорости разработки это позволительно. Так как основной интерес использования данного алгоритма приходится на туристов, а, следовательно, и мобильные устройства, исходя из этих выводов стоит переносить вычисления в облако, избавляя маломощные устройства от больших нагрузок.

Примеры решений

Покажем некоторые конкретные примеры решений.

Время измеряется в секундах.

Пример 1.

Элементарный граф в пять вершин и с ограничением по времени $T_{max} = 4$ (рис. 3). Можно увидеть, что алгоритм выбирает оптимальный маршрут с учетом временных ограничений.

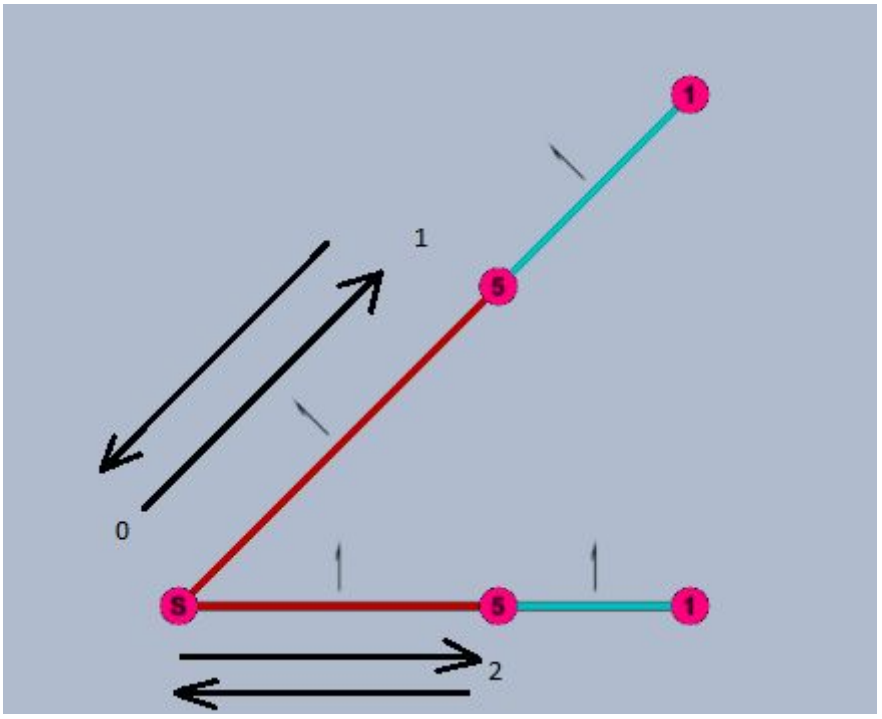

рис. 3

Пример 2.

Циклический граф (рис. 4.1) с ограничением по времени Tmax = 5, показывающий возможность алгоритма выбрать лучший маршрут через неиспользованную ветвь на обратном пути. Таким образом, по сравнению с Примером 1, алгоритм не возвращался в стартовую вершину, а обошел граф через каждую вершину, набрав максимальный интерес.

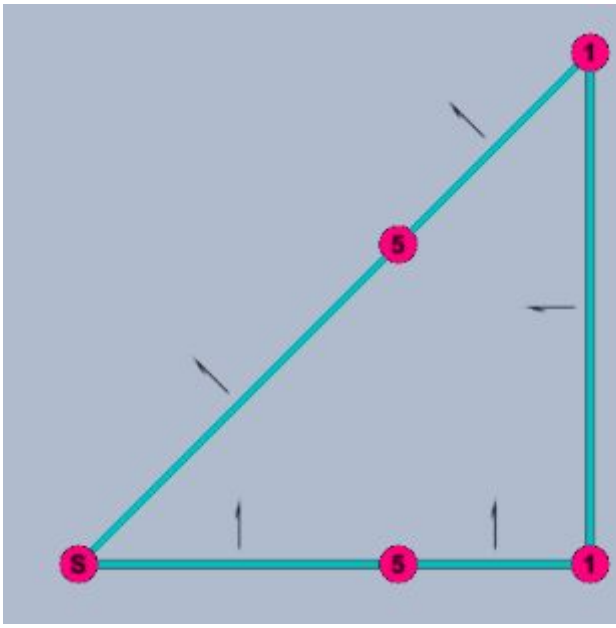

рис. 4.1

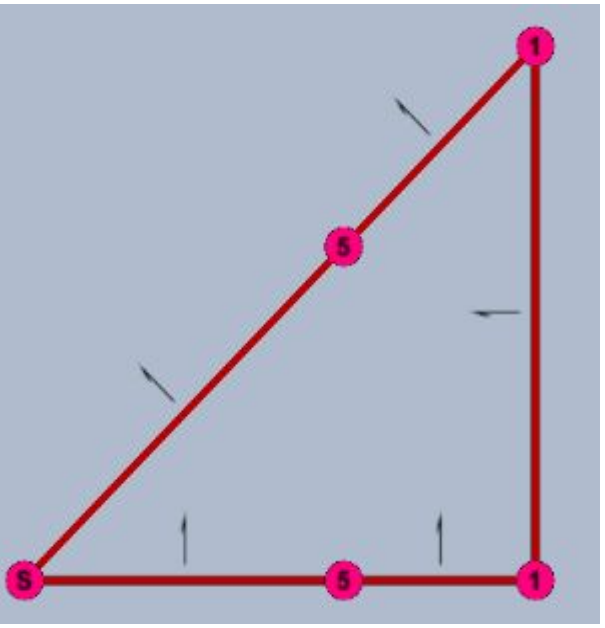

рис. 4.2

Пример 3.

Граф с необходимостью выбрать правильный путь, на рис. 5 видно, что алгоритм не отдает приоритет ребрам с наименьшим весом и может добавить в путь вершины наихудшие на данный момент.

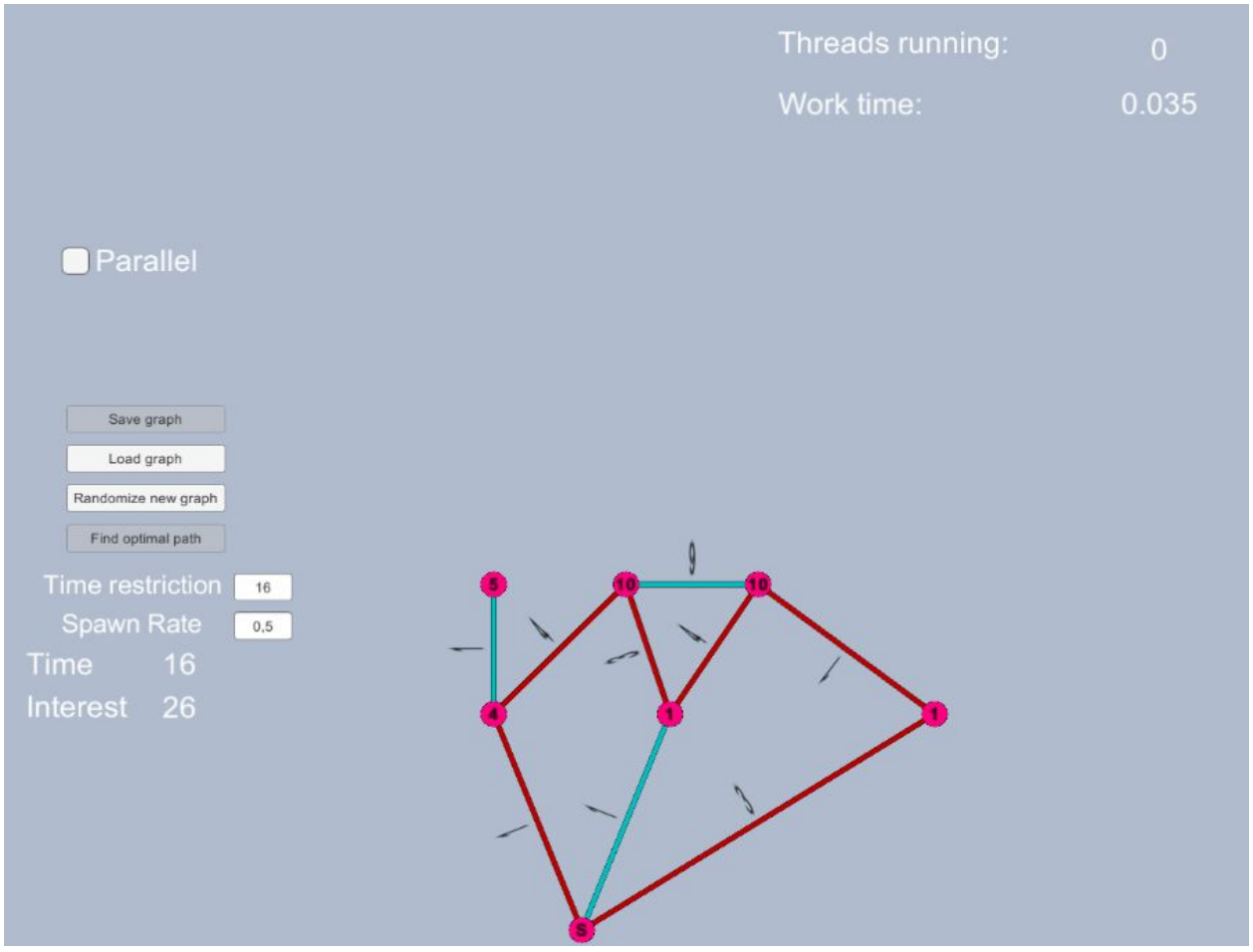

рис. 5

Пример 4.

В данном примере рассматривается работа алгоритма на сгенерированном графе с вероятностью связи вершины с вершиной в 0.4.

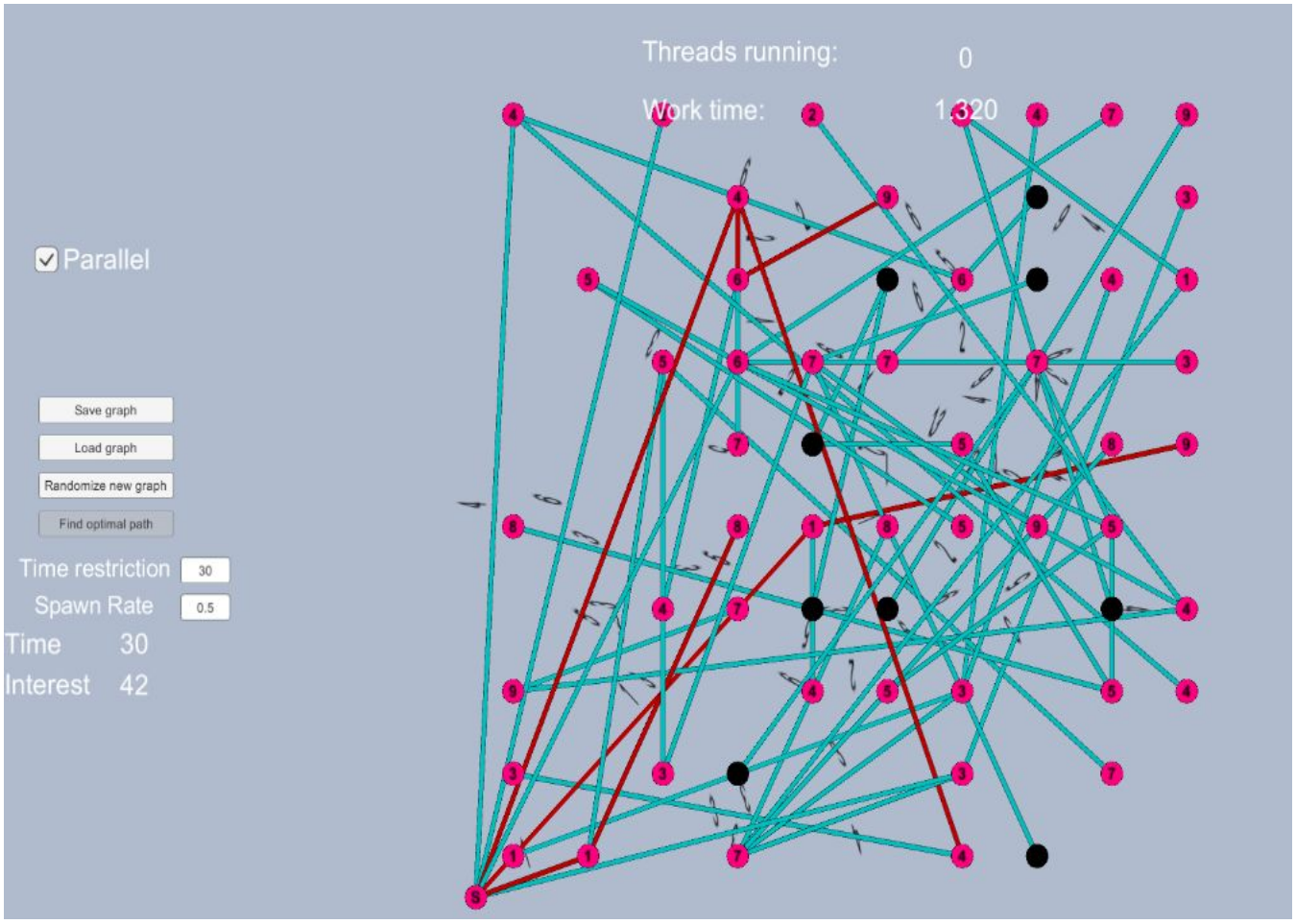

рис. 6

(Черным отмечены недостижимые из стартовой за временное ограничение вершины)

На рис. 6 можно увидеть, что алгоритм в параллельном режиме успешно завершил свою работу за время 1.32 секунды. Последовательный режим выдает такой же результат за 1.52 секунды. Так как граф небольшой, ускорение параллельного режима можно назвать незначительным, из-за того, что в замер времени входит подготовительный этап, необходимый для работы пула потоков,.

Выводы

Были рассмотрены различные подходы к решению, которые можно взять за решение поставленной задачи, такие как: поиск в глубину, жадные алгоритмы, метод ветвей и границ. Разработан собственный алгоритм, реализующий поиск оптимизирующего маршрута за приемлемое время, позволяющий выполнять вычисления как в параллельном режиме, так и в последовательном. Алгоритм использует эвристику, позволяющую сократить количество рассматриваемых вариантов. Проведен анализ полученных результатов. Была реализована визуализация процесса для наглядности и анализа. Программный комплекс предоставляет возможность генерировать, сохранять и загружать рандомизированные графы, с помощью которых можно легко протестировать работу алгоритма. Также работа позволяет выводить статистику по текущему решению и сравнивать его с максимальным на графе.

Визуальный интерфейс и возможность загрузки произвольного и удовлетворяющего условиям графа позволяет без дополнительных усилий рассчитать на нем лучший маршрут.

К недостаткам алгоритма можно отнести значительное возрастание времени работы относительно количества вершин, ребер между ними и ограничения по времени, однако при использовании программы для реальных прогулок необходимый маршрут будет строиться за приемлемое время.

Заключение

Построение маршрута для прогулок всегда было актуальным в среде туризма, а с развитием информационных технологий алгоритмы, подобные описанным в данной работе, будут становиться лишь востребованней.

Разработанная архитектура далека от идеала, но остается открытой для модификаций. Среди возможных улучшений можно выделить следующие:

- Перенос вычислений в облачную архитектуру: разгрузка вычислительных мощностей мобильных устройств всегда будет хорошо воспринята со стороны пользователей;
- Улучшение работы алгоритма с эвристикой: на текущий момент алгоритм использует эвристику лишь для сокращения числа рассматриваемых вариантов, но в перспективе правильно выбранная эвристика позволит достичь значительного возрастания производительности;
- Улучшение синхронизации потоков: использование функциональности ThreadPool предоставляет приемлемое взаимодействие потоков, но написанная специально под нашу задачу функциональность должна сократить время простоя потоков в блокировке и улучшить обмен информацией между ветвями.

Самым перспективным улучшением работы алгоритма является предварительная обработка входных данных: изначально удаленные излишние переходы между вершинами, склеивание накладывающихся вершин друг на друга. В идеале входные данные должны быть без лишних переходов и тогда алгоритм всегда будет отрабатывать за лучшее время.

Список использованных источников

1. J. Harris, J.L. Hirst, and M. Mossinghoff. Combinatorics and Graph Theory. Springer, 2009.
2. Задача коммивояжера и методы ее решения. http://mech.math.msu.su/~shvetz/54/inf/perl-problems/chCommisVoyageur.xhtml
3. The Single Vehicle Pickup and Delivery Problem with Time Windows: Intelligent Operators for Heuristic and Metaheuristic Algorithms. https://users.cs.cf.ac.uk/C.L.Mumford/papers/M_Hosny.pdf
4. Базовые алгоритмы нахождения кратчайших путей во взвешенных графах https://habrahabr.ru/post/119158/
5. Simulated annealing https://habrahabr.ru/post/209610/
6. ThreadPool Class msdn documentation https://msdn.microsoft.com/ru-ru/library/system.threading.threadpool(v=vs.110).aspx
7. Bellman-Ford algorithm https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm.
8. A Tabu Search Heuristic for the Vehicle Routing Problem http://www.jstor.org/stable/2661622
9. TABU SEARCH FUNDAMENTALS AND USES by Fred Glover US West Chair in Systems Science Graduate School of Business https://pdfs.semanticscholar.org/e8be/8666bfe90eda68f7a5fe4723c7f8e1d1c9c3.pdf
10. Tabu search https://en.wikipedia.org/wiki/Tabu_search

11. Ant colony optimization

    https://en.wikipedia.org/wiki/Ant_colony_optimization_algorithms

Приложение

Данные тестов:

0.9	0.7	0.5	0.3	0.1
Послед.	Послед.	Послед.	Послед.	Послед.
1. 0.07	1. 1.778	1. 0.47	1. 13.6	1. 11.03
2. 0.012	2. 0.286	2. 1.798	2. 13.975	2. 17.67
3. 0.704	3. 0.066	3. 0.029	3. 1.3	3. 0.738
4. 0.005	4. 0.203	4. 0.586	4. 0.509	4. 0.092
5. 0.112	5. 0.052	5. 4.7	5. 1.314	5. 0.074
6. 0.052	6. 0.088	6. 0.97	6. 6.07	6. 4.100
7. 0.014	7. 0.339	7. 0.668	7. 2.052	7. 9.054
8. 0.04	8. 0.092	8. 0.369	8. 3.486	8. 3.882
9. 0.016	9. 0.038	9. 0.326	9. 0.192	9. 13.429
10. 0.004	10. 0.142	10. 0.477	10. 0.322	10. 1.096
Параллел.	Параллел.	Параллел.	Параллел.	Параллел.
1. 0.043	1. 1.26	1. 0.056	1. 12.5	1. 7.822
2. 0.009	2. 0.198	2. 1.035	2. 14.5	2. 11.25
3. 0.406	3. 0.044	3. 0.024	3. 0.586	3. 0.454
4. 0.001	4. 0.183	4. 0.5	4. 0.331	4. 0.048
5. 0.05	5. 0.024	5. 3.6	5. 0.738	5. 0.064
6. 0.029	6. 0.087	6. 0.622	6. 4.91	6. 3.026
7. 0.011	7. 0.245	7. 0.367	7. 1.649	7. 7.755
8. 0.021	8. 0.053	8. 0.354	8. 1.541	8. 2.880
9. 0.002	9. 0.023	9. 0.172	9. 0.101	9. 9.499
10. 0.003	10. 0.089	10. 0.262	10. 0.277	10. 0.854